Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC-CAL Dataflow Programs

Jani Boutellier, Mickaël Raulet, Olli Silvén

Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC-CAL Dataflow Programs

Jani Boutellier · Mickaël Raulet · Olli Silvén

Abstract RVC-CAL is an actor-based dataflow language that enables concurrent, modular and portable description of signal processing algorithms. RVC-CAL programs can be compiled to implementation languages such as C/C++ and VHDL for producing software or hardware implementations.

This paper presents a methodology for automatic discovery of piecewise-deterministic (quasi-static) execution schedules for RVC-CAL program software implementations. Quasi-static scheduling moves computational burden from the implementable run-time system to design-time compilation and thus enables making signal processing systems more efficient.

The presented methodology divides the RVC-CAL program into segments and hierarchically detects quasi-static behavior from each segment: first at the level of actors and later at the level of the whole segment. Finally, a code generator creates a quasi-statically scheduled version of the program.

The impact of segment based quasi-static scheduling is demonstrated by applying the methodology to several RVC-CAL programs that execute up to 58% faster after applying the presented methodology.

Keywords Scheduling · Signal processing · Data flow analysis

1 Introduction

Since decades, data flow models of computation have been widely used for describing signal processing applications. One of the recently popularized data flow languages is called RVC-CAL [3], which is a restricted version of the CAL [6] language.

J. Boutellier (contact) · O. Silvén
Department of Computer Science and Engineering,
90014 University of Oulu, Finland
e-mail: jani.boutellier@ee.oulu.fi

O. Silvén
e-mail: olli.silven@ee.oulu.fi

M. Raulet
IETR/INSA Rennes, 35043, Rennes, France
e-mail: mickael.raulet@insa-rennes.fr

The model of computation under the RVC-CAL language is dynamic by nature, which makes it suitable for describing signal processing algorithms that have data-dependent behavior [1]. However, this expressiveness of the language also greatly disables traditional data flow analysis approaches.

One of the key features that make traditional Synchronous Data Flow [7] models attractive is that they allow static compile-time scheduling, which removes the need for costly run-time schedulers. For programs written in the dynamic RVC-CAL language it is generally not possible to find completely static execution schedules. Instead, piecewise-static (quasi-static) schedules can be discovered. The problem is challenging [5], and to this date there has been no general, automated solution.

This paper presents an automatic, hierarchical methodology for discovering quasi-static schedules for RVC-CAL programs. We present results where programs have been re-compiled with quasi-static schedules and thus speeded up even 58%.

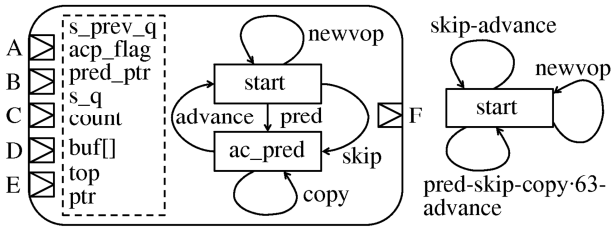

Fig. 1 Left: the actor "IAP" from the "RVC" program. Right: the same actor simplified by the proposed methodology.

2 The Computation Model under RVC-CAL

RVC-CAL is an *actor*-based data flow language. Actors are entities that perform computations and exchange data with each other explicitly over FIFO channels [7]. An RVC-CAL program contains at least one actor, or a *network* of interconnected actors. In RVC-CAL there are no global variables; data between actors is exchanged over FIFOs as entities called *tokens*.

Fig. 1 shows an RVC-CAL actor. The actor has five input ports and one output port, each connected to a FIFO. The actor is controlled by a Finite State Machine (FSM) with two states: *start* and *ac_pred*. Computations are performed in state transitions called *actions*. Data-dependent execution can be seen in actions *skip* and *pred*: depending on the token value of input port D, either *skip* or *pred* is executed. If the execution of an action depends on the token value of a port, the action is called a *data-dependent action* and the port is a *data dependent port*.

An action can execute when it has enough input tokens (a number called *token rate* [5]), there is sufficient space in the output FIFOs and the *guard* condition is met. Guard conditions can be related to variable or token values [2]. The actor in Fig. 1 has eight variables, of which one is an array.

2.1 Terminology

Having an RVC-CAL program *n* that consists of *N* actors, *segment s* is a subset of *n* that contains 1 to *N* actors. Each actor of *n* must belong to exactly one segment *s*. The set of all segments is called a *segmentation*, *S*, and it contains 1 to *N* different segments. A segment has a set of output ports s_o and a set of input ports s_i that belong to the actors within *s*.

An actor *a* belonging to segment *s* is called a *back-edge actor* if at least one of its output ports belongs to s_o. Respectively, an actor *a* of segment *s* is called a *front-edge actor* if at least one of its input ports belongs to s_i.

A *buffering actor* is an actor that does not immediately produce tokens to its output ports after consuming tokens from its input ports; instead, data is stored inside the actor and output during a later invocation [2].

An *action chain*, *c*, is a sequence of actions. The aim of our methodology is to express segments as a set of one or more deterministic action chains.

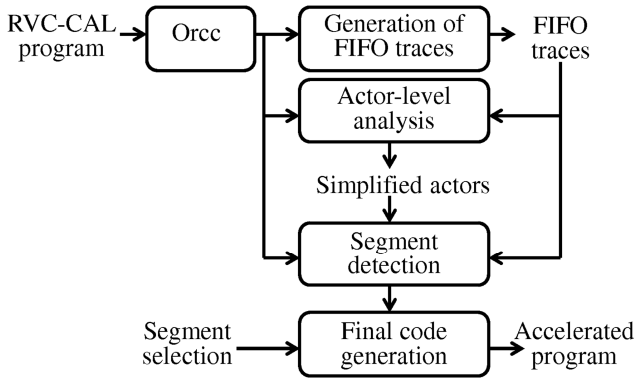

Fig. 2 The steps of discovering quasi-static segments.

3 Related Work

Gu et al. [1] present a framework for detecting statically schedulable regions from CAL programs, which is similar to this work: the statically schedulable regions of [1] are analogous to action chains. The difference to the work presented here is that in the work of Gu et al. there is no concept of segments; each actor may belong to several different static regions.

Ersfolk et al. [2] present a model-checking based approach to discover execution schedules for parts of RVC-CAL networks. The approach of Ersfolk et al. is similar to the work presented in this paper: both assume that there is a set of RVC-CAL actors (a segment) for which a set of alternative execution sequences (action chains) are determined. However, based on the experiments presented in [2] it is not possible to numerically compare their approach to the one presented in this paper.

Boutellier et al. [4] present an approach to analyze RVC-CAL networks and derive quasi-static execution schedules. Unlike the presented work, the approach in [4] considers the whole RVC-CAL program and attempts to parameterize its behavior with one dynamic signal (FIFO connection). For programs with a single dynamic signal the approach works well, but for networks for which this assumption does not hold, the gain is rather modest.

The work presented here allows an arbitrary number of dynamic signals in the program and enables deriving quasi-static schedules for an arbitrary number of non-overlapping segments.

4 The Proposed Approach

The aim of our methodology is to automatically detect quasi-statically schedulable segments from RVC-CAL programs. When such a segment has been detected, the actors within that segment can be replaced with a set of deterministic action chains that can be executed instead of actor state machines.

If the behavior of a segment can be fully described with a set of action chains, it is possible to generate a simple run-time scheduler for that segment, which based on a few conditions executes the appropriate action chain instead of repeatedly trying to execute individual actions from separate actors.

Practically our methodology has been implemented with dynamic code analysis [4]: the program is equipped with instrumentation code that allows observing its run-time behavior. Code generation for all steps is done by Orcc (http://orcc.sf.net).

Below, the individual steps of the quasi-static segment discovery process are described in detail (overview in Fig. 2).

4.1 Generation of FIFO Traces

In the first step of our methodology, the RVC-CAL program is executed with training data and the tokens flowing within each FIFO buffer of the RVC-CAL actor network are recorded as *FIFO traces*. The training data must be sufficiently diverse to exhibit all behaviors that the actors are capable of performing.

4.2 Actor-Level Analysis

In the second step the FIFO traces are used to execute and analyze each RVC-CAL actor individually, in isolation. The aim of Step 2 is to discover any possible repetitive and deterministic behavior that an actor may have. With knowledge of deterministic behavior it is possible to write a simplified version of each actor for easier analysis at segment level. An example result of this is shown in Fig. 1: the simplified actor has one state and three action chains of lengths 2, 1 and 66 actions. Below, the actor-level analysis is explained in three substeps.

4.2.1 Substep 1 - Detecting Action Chains

Each actor is executed in isolation with the FIFO traces acquired in Step 1; as a result an actor *execution trace* is acquired. The execution trace holds the sequence of actions that were executed

as a result of the FIFO traces used, along with full actor state at the moment of executing each action.

Deterministic actor behavior is discovered by analyzing the execution trace and detecting action chains (sequences of actions) that always follow each other. The detection is based on a) execution of data dependent actions, b) visited FSM states or c) momentary variable values. A few examples follow.

For actors that have data dependent actions, a new action chain is determined to start whenever a data dependent action execution is detected in the execution trace.

If the actor does not have any data dependent actions, but has multiple FSM states, the execution trace is split into action chains based on visited FSM states. The execution trace is analyzed separately for each FSM state $x \in X$ (where X is the set of all FSM states of the actor). When the analysis is running for state x, a new action chain is determined to start whenever the execution trace visits FSM state x.

Finally, if the actor has no data dependent actions and only one FSM state, the execution trace is split into action chains based on variable values: a new action chain is determined to start whenever the variables of the actor have certain value v. v is a vector that contains all guard-related variables of that actor.

If the execution trace cannot be split into action chains based on any of these conditions, we call the actor *trivial*.

Nondeterministic actors express behavior that cannot be modeled quasi-statically due to, for example, a data-dependent number of loop iterations.

4.2.2 Substep 2 - Verifying and Pruning Action Chains

Substep 1 also produces action chains that in reality are not deterministic. These unfit action chains are eliminated by verifying each action chain c back against the actor's execution trace t. At each execution trace position t_p that exhibits the first action of an action chain c_0, a comparison is done: all actions c_i (i = 0, 1, 2, … L; where L is the length of c) must match t_{p+i}.

If $c_i \neq t_{p+i}$, c is split at position i, into two new chains d and e such that $c = d||e$. The new chains are added to the end of the action chain list and are verified similarly later on.

After verification, unnecessary action chains are discarded by rebuilding the execution trace of the actor from beginning to end by using only the action chains. Chains not needed for rebuilding the trace are discarded.

4.2.3 Substep 3 - Constructing Simplified Actors

If the procedure of reconstructing the execution trace succeeds, a new, simplified actor (see Fig. 1) can be built based on the action chains. The new actor may have fewer states in the FSM and can perform more computations with one invocation. Naturally, there are actors for which a simplified version cannot be constructed. In these cases, the original actor is used in the later steps of the presented methodology. This is the case for trivial actors and *nondeterministic* actors. Nondeterministic actors exhibit such complex behavior that they cannot be modeled with the approaches described in Section 4.2.1.

4.3 Discovering Segments

At the beginning of the third step of our methodology, the RVC-CAL program is rebuilt such that actors are replaced by their simplified versions whenever possible.

Next, our methodology starts detecting segments that allow quasi-static schedules to be generated. The following rules are used to detect suitable segments: a) A segment consisting of more than one actor may not contain nondeterministic actors. b) A segment may not have data dependent input ports that originate from more than one source port. c) A trivial actor may not be a front-edge actor of the segment. d) A buffering actor may only be a back-edge actor of a segment. e) Every actor a in segment s must be the predecessor or the successor (in the sense of directed graphs) of every other actor in segment s. f) No segment s may have both input ports s_i and output ports s_o that connect to another segment r.

The discovery of segments is currently implemented by random search: segments are generated randomly, after which conditions a) to f) are checked. If a segment meets all the conditions the segment is eligible for detecting action chains at segment level. Detection of action chains at segment level is performed by executing the segment in isolation using FIFO traces, similar to what was described in Section 4.2.1.

To avoid the segment level action chains from becoming overtly long, the isolated execution is restricted by limiting the number of tokens the segment may consume on one execution. For each s_i of segment s, the maximum allowed consumption is equal to the token rate of the actor port.

Each time before the segment is allowed to execute, the initial state of the segment, *signature* [4], is recorded. The signature consists of 1) value of the first token in each s_i that is data dependent, 2) the FSM state of each actor in the segment, 3) the values of variables within each actor of the segment, 4) the number of tokens on each FIFO of s which does not belong to s_i or s_o. In the code generation step, the signature is used to build a selection mechanism to arbitrate between different action chains.

4.4 Final Code Generation

In general, the segment discovery process produces several non-unique segmentations, out of which one must be chosen for the final implementation of the RVC-CAL program. Currently, this selection is done manually.

After the designer has selected the segmentation, the final executable code is automatically generated. For segments that can be expressed as a limited number of deterministic action chains, the code generator automatically generates quasi-static execution schedules and a suitable arbitration mechanism to decide at run-time which action chain to execute. The code generator is similar to the one used in [4].

An example that shows the efficiency of the proposed approach: for the "mc" segment (Fig. 3), the code generator instantiates seven different action chains that consist of up to 380 actions each. These seven action chains fully capture the behavior of the four actors in the segment. The automatically

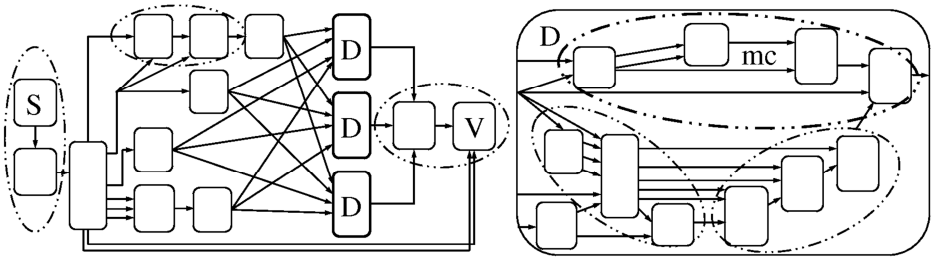

Fig. 3 The "RVC" actor network. *S* and *V* are source and sink actors. *D* is a hierarchical actor existing as three instances. The actor network inside D is on the right. The "mc"-oval is the motion compensation loop selected for code generation; the rest of the ovals show some other detected segments.

generated run-time scheduler is then able to arbitrate between the different action chains based on a single variable value.

For segments that do not exhibit such deterministic behavior, calls to the default actor scheduler are used.

5 Experiments

Experiments were performed on three RVC-CAL programs to evaluate the proposed methodology. Each of the programs was an MPEG-4 Simple Profile video decoder described in a different fashion. A video sequence named "l.m4v" of resolution 720x480 and length of 5 frames was used as training data for applying the methodology to each of the programs.

Table 1 shows the performance of the "RVC" program in frames/sec (fps) before and after applying our methodology. For the segmented version, three automatically detected segments (see Fig. 3) were selected for generating the improved program. These results show that even with a small amount of training data, the behavior of the program can be analyzed and improved correctly and the speedup is considerable even though the quasi-static code generation is very basic at the moment.

On the workstation platform that was used for experiments, the use of simplified actors (Section 4.2.3) alone produced no speedup, but helped in the discovery of quasi-static schedules at segment level. On an embedded platform with small buffer sizes the results might be different, but this was not experimented.

Table 2 shows the number of actors and a performance summary for all three programs. The number of automatically detected segments *with two or more actors* is shown as well. The performance numbers show the average performance of all 5 video sequences (l.m4v, m1.m4v m2.m4v, …) over 500 frames.

The experimentation platform was Ubuntu Linux 11.10 running on Windows 7 over VMWare player. The processor was Intel Core 2 Duo E8500 and the compiler GCC 4.6.1.

6 Conclusions

In this paper we have presented a methodology for automatic discovery of quasi-static segments for RVC-CAL programs. The methodology is based on hierarchical process that has in this work been implemented via dynamic code analysis. Results show the considerable performance gain that can be achieved. Studying the implementation of the presented methodology by static analysis [2] and creating a more sophisticated quasi-static code generator remain as future goals.

Table 1 Performance of program "RVC" in frames per second, as well as the percentage of macroblocks having texture data and those that are interpolated.

Video seq.	Int. MBs	Tex. MBs	Ordinary	Segmented
m1.m4v	91.6%	21.4%	13.5	21.5
m2.m4v	76.3%	47.9%	12.1	17.8
s1.m4v	94.0%	25.1%	12.9	20.2
s2.m4v	84.2%	39.3%	12.7	19.1

Table 2 Number of detected segments with two or more actors.

Program	Actors	Segments	Fps/O	Fps/S
MPEG-4 "MVG"	21	45	20.9	25.1
MPEG-4 "RVC"	45	39	13.1	20.7
MPEG-4 "Serial"	25	28	17.6	22.8

Acknowledgements

This research has been partially funded by the DORADO project of the Academy of Finland.

References

[1] Gu, R., Janneck, J. W., Raulet, M., and Bhattacharyya, S. S. (2011). Exploiting Statically Schedulable Regions in Dataflow Programs. *Journal of Signal Processing Systems*, 63(1):129-142.

[2] Ersfolk, J., Roquier, G., Jokhio, F., Lilius, J. and Mattavelli, M. (2011). Scheduling of Dynamic Dataflow Programs with Model Checking. Proceedings of the 2011 IEEE Workshop on Signal Processing Systems (SiPS), 37-42.

[3] Janneck, J. W., Mattavelli, M., Raulet, M., and Wipliez, M. (2010) Reconfigurable Video Coding: a Stream Programming Approach to the Specification of New Video Coding Standards. Proceedings of ACM Multimedia Systems 2010, 223-234.

[4] Boutellier, J., Silvén, O. and Raulet, M. (2011). Scheduling of CAL actor networks based on dynamic code analysis. Proceedings of the 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 1609-1612.

[5] Boutellier, J., Lucarz, C., Lafond, S., Martin Gomez V. and Mattavelli M. (2011) Quasi-Static Scheduling of CAL Actor Networks for Reconfigurable Video Coding. *Journal of Signal Processing Systems*, 63(2):191-202.

[6] Eker, J. and Janneck, J. (2003) CAL Language Report. Technical Report UCB/ERL M03/48, UC Berkeley.

[7] Lee, E. A., and Messerschmitt, D. G. (1987) Synchronous Data Flow. *Proceedings of the IEEE*, 75(9), 1235-1245.